Validation of Object-Oriented Concurrent Designs by Model Checking*

Klaus Schneider, Michaela Huhn, and George Logothetis

University of Karlsruhe, Department of Computer Science
Institute for Computer Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
{schneide,huhn,logo}@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/

1 Introduction

Reusability and evolutivity are important advantages to introduce object-oriented modeling and design also for embedded systems [1,2]. For this domain, one of the most important issues is to validate the interactions of a set of objects with concurrent methods. We apply model checking (see [3] for a survey) for the *systematic debugging of concurrent designs* to detect errors in the behavior and interactions of the object community. As we assume a fixed finite maximal number of objects and also finite data types, we can only show the correctness for finite instances and detect only errors that appear in such a finite setting. Nevertheless, the approach is useful for embedded systems, where the system's size is limited by strong hardware constraints. Moreover, we claim that most errors in the concurrent behavior already occur with a small number of components. To handle larger designs, we emphasize that it is often obvious that several attributes of an object do not affect the property of interest. Thus, there is no need to model the objects completely. This obvious abstraction leads to significantly better results because the resulting models are smaller. More sophisticated abstractions can be found e.g. in [4,5]. In the next section, we briefly explain how to derive in general a finite state system from an object-oriented concurrent design. Then, we illustrate the method by a case study taken from [6].

2 From Concurrent Objects to Finite State Machines

We assume a class $\mathcal{C}$ with attributes $a_1, \ldots, a_n$ of types $\alpha_1, \ldots, \alpha_n$, respectively, and methods $\tau_0, \ldots, \tau_m$ to manipulate the attributes (there may be constructor and destructor methods). For an object $\mathcal{O}$ of class $\mathcal{C}$, we denote the method τ_i invoked for $\mathcal{O}$ by $\mathcal{O}.\tau_i$ and the value of the attribute a_i of $\mathcal{O}$ by $\mathcal{O}.a_i$.

We are not interested in inheritance or typing issues, but in the concurrent behavior: the methods $\mathcal{O}.\tau_i$ are implemented as threads, i.e., they may be invoked in parallel. Their execution may be interleaved or truely concurrent. The

* This work has been financed by the DFG priority program 'Design and Design Methodology of Embedded Systems'.

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 360–365, 1999.

methods work on the same memory (namely the attributes of $\mathcal{O}$). The methods $\mathcal{O}.\tau_i$ are not necessarily atomic, instead they consist of a sequence of atomic operations. Hence, a method $\mathcal{O}.\tau_i$ may be suspended, aborted or interrupted by another method $\mathcal{O}.\tau_j$. A major concern is that the concurrent execution does not lead to inconsistencies or runtime faults. This problem is nontrivial since concurrent threads may modify the same attributes $\mathcal{O}.a_i$ or interrupt such modifications before they are completed. For our finite state machine abstraction, three different kinds of abstractions have to be applied systematically:

- *Infinite data types* are abstracted to a finite data domain. E.g., integers are mapped to bitvectors of a certain length n.
- While *objects may come and go dynamically*, the finite state machine model requires to fix a maximal finite number of objects that invariantly exist from the beginning. Hence, we allocate for each class a maximal finite number of objects in advance, and model construction and destruction such that at each point of time at most the maximal number of these objects are in use.
- All methods $\mathcal{O}_j.\tau_i$ are modeled as finite state machines $\mathcal{A}_{j,i}$ that may interact with each other. As the entire system must be finite-state, we have to *restrict the maximal number of methods that may run in parallel at a point of time*. Similar to the construction and destruction of objects, we can however model that threads are dynamically started, suspended, aborted, or that they terminate.

Due to the description level and the properties of interest, a granularity of atomicity has to be chosen. For instance, for a system design given in Java, it is reasonable to assume Java statements as atomic. Also, the treatment of write-clashes has to be modeled: One could either nondeterministically select a value or determine the value by a resolution function as common in many concurrent languages. Our model is able to cope with any of these solutions.

In contrast to other formal approaches to object-oriented designs, we consider methods as non-atomic and allow the concurrent methods running on the same object. Hence, our model is closer to practical implementations in C++ or Java and even takes the thread management of the operating system into account. In particular, we are interested in whether the design is robust wrt. the suspension and activation of threads. We emphasize that the construction of the finite-state model can be done automatically, when the maximal number of concurrent threads and concurrently existing objects are fixed, and the mapping from infinite data types to finite ones is given.

3 Interrupt-Transparent Lists

We now present a case study taken from an embedded operating system kernel [6]. The objects are single-linked lists. The methods to modify the list may be interrupted at any time. Therefore, naive sequential list operations can not be used, since the list could become inconsistent (in particular, some items get

```
class Chain
 {public:Chain* next};

class Cargo:public Chain
 {public:
   Chain* tail;
   Cargo();
   void enqueue(Chain* item);
 };

Cargo::Cargo()
 {next = 0;
  tail = (Chain*) this;}
```

```
void Cargo::enqueue(Chain* item)
  {Chain *last;
   Chain *curr;
 s1 : item->next = 0;
 s2 : last = tail;
 s3 : tail = item;
 s4 : if (last->next)
 s5 :  { curr = last;
 s6 :    while (curr->next)
 s7 :       curr = curr->next;
 s8 :    last = curr;
       }
 s9 : last->next = item;
  }
```

Figure1. Implementation of Interrupt-Transparent Lists

lost). Due to the lack of space, we only consider the enqueue method. C++ implementations are given in figure 1.
`Cargo`-objects consist of two pointers: `next` points to the first list element, and `tail` points to the last one. The `Cargo` constructor assigns `next` to 0, and `tail` to the current object. To ensure robustness against interrupts, we need a sophisticated implementation of the enqueue method due to [6]. Given a `Chain` object `item`, `enqueue` resets the `next`-pointer of the argument (`s1`), and stores the current end of the list (`s2`). In line `s3`, it assigns the `tail`-pointer to the argument. If the thread is not interrupted `last->next` must be 0 at `s3`. Otherwise an interrupting enqueue thread may have inserted some elements. Then, the current end of the list has to be found (loop `s5`, ..., `s8`). Finally, the argument is added to the list (`s9`).

Modeling Lists of Bounded Length. To derive a finite-state model, we consider a maximal number of n enqueue threads $E_1, \ldots, E_n$ manipulating in an interleaving execution one `Cargo` object. Analogous to the C++ code, we abstract from the items and simply enumerate them: E_i inserts item i. The `tail` pointer of the `Cargo` object is represented by the variable *tail*, the `next` pointer by $pt[0]$. The C++ expressions `a->next` and `a->next->next` correspond in our model with the terms $pt[a]$ and $pt[pt[i]]$, respectively. We start with the initial values $pt[0] = 0$ and $tail = 0$.

type enum *Item:* $1, \ldots, n$; var pt : array *Address* of *Address*;
type enum *Address:* $0, \ldots, n$; var *tail : Address;*

Modeling the Enqueue Threads. Each E_i is modeled as a finite state machine that manipulates two local variables $last_i$ and $curr_i$ of type *Address*. The states directly correspond C++ code of figure 1. States s_0 and s_{10} are added to model that E_i has not yet started or has already terminated.

s_0 :
s_1 : $pt[i] := 0$
s_2 : $last_i := tail$
s_3 : $tail := i$
s_4 : if $pt[last_i] = 0$ goto s_9 else goto s_5
s_5 : $curr_i := last_i$
s_6 : if $pt[curr_i] = 0$ goto s_8 else goto s_7
s_7 : $curr_i := pt[curr_i]$; goto s_6
s_8 : $last_i := curr_i$
s_8 : $last_i := curr_i$
s_9 : $pt[last_i] = i$
s_{10} :

As the threads $E_1, \ldots E_n$ run in an interleaved manner, at most one thread manipulates the `Cargo` object at each point of time. Without loss of generality, we assume that E_i starts before E_j iff $i < j$. If E_j starts when E_i is still running, E_i will be interrupted, since E_j has a higher priority due to the interruption. The interruption is modeled by the following signals: $run_i := \bigvee_{k=1}^{9} E_i.s_k$, $end_i := E_i.s_{10}$, $ac_i := \bigwedge_{k=i+1}^{n} \neg run_k$, and $perm_i := \bigwedge_{k=1}^{i-1} start_k$. run_i holds iff thread E_i is currently running. end_i holds iff thread E_i has terminated. ac_i indicates that E_i is activated, i.e., no thread E_k with higher priority ($i < k$) is currently running. Finally, $perm_i$ implements the priorities, i.e. E_i may only start if all E_j with $j < i$ have started before. Using these control signals, we obtain the state transition diagram as given in figure 2.

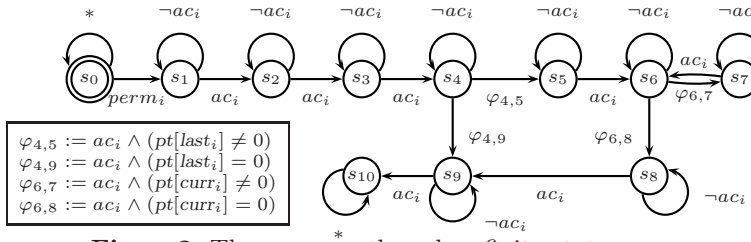

Figure2. The enqueue thread as finite state process

Properties to be Checked The properties we checked using McMillan's SMV system[1] are as follows:

S_1**:** All items are enqueued: $\mathsf{G}\left[(\bigwedge_{i=1}^{n} end_i) \rightarrow \left(\bigwedge_{i=1}^{n} \bigvee_{j=0}^{n} pt[j] = i\right)\right]$
S_2**:** All items are enqueued at most once:

$$\mathsf{G}\left[\left(\bigwedge_{i=1}^{n} end_i\right) \rightarrow \left(\bigwedge_{i=1}^{n} \bigvee_{j=0}^{n} pt[j] = i \rightarrow \bigwedge_{k=0}^{n} pt[k] = i \rightarrow k = j\right)\right]$$

S_3**:** Absence of deadlocks: $\mathsf{G}\,[\bigwedge_{i=1}^{n} start_i \wedge perm_i \rightarrow \mathsf{F} end_i]$
S_4**:** The threads are started in order $E_1, E_2, \ldots E_n$: $\mathsf{G} \bigwedge_{i=1}^{n-1} E_i.s_0 \rightarrow \bigwedge_{j=i+1}^{n} E_j.s_0$

[1] Available from http://www-cad.eecs.berkeley.edu/~kenmcmil/

S_5: If $E_1, \ldots E_k$ have terminated *before* any of the threads $E_{k+1}, \ldots E_n$ has started, the first k elements have been enqueued:

$$\mathsf{G} \bigwedge_{k=1}^{n-1} \left[\left(\bigwedge_{i=1}^{k} end_i \right) \wedge \left(\bigwedge_{i=k+1}^{n} E_i.s_0 \right) \rightarrow \bigwedge_{j=0}^{k} \bigvee_{i=0}^{k} pt[i] = j \right]$$

$\bigwedge_{j=0}^{k} \bigvee_{i=0}^{k} pt[i] = j$ means $pt[0], \ldots, pt[k]$ is a permutation of $0, \ldots, k$, which means that each element of $0, \ldots, k$ is contained in the list.

S_6: If $E_1, \ldots E_k$ have terminated *before* any of the threads $E_{k+1}, \ldots E_n$ has started, then the elements $1, \ldots, k$ will occur before the elements $k+1, \ldots, n$ in the final list. Moreover, the k-prefix of the list is stable from the moment where all $E_1, \ldots E_k$ have terminated:

$$\mathsf{G} \bigwedge_{k=1}^{n-1} \left[\left(\bigwedge_{i=1}^{k} end_i \right) \wedge \left(\bigwedge_{i=k+1}^{n} E_i.s_0 \right) \rightarrow \mathsf{G} \bigwedge_{j=1}^{k} \bigvee_{i=0}^{k} pt[i] = j \right]$$

The experiments were made on a Intel Pentium with 450 MHz and 512 MByte main memory under Linux. For the results marked with *, we had to use the dynamic sifting option of SMV which increases the runtime due to optimization of the BDD sizes. All properties were checked with moderate resources, besides the deadlock detection.

Property/threads	Runtime [seconds]	BDD Nodes
$S_1/4$	7.68	584141
$S_1/5$	95.29	7682664
$S_2/4$	7.64	584580
$S_2/5$	95.97	7683911
$S_3/4$	3 40.10	1897469
$S_3/5^*$	20383.78	30593433

Property/threads	Runtime [seconds]	BDD Nodes
$S_4/4$	21.20	1976314
$S_4/5^*$	554.55	823411
$S_5/4$	8.19	599318
$S_5/5$	105.75	9592257
$S_6/4$	17.69	1216240
$S_6/5$	205.25	10724191

Acknowledgment

We thank Prof. Schröder-Preikschat, Ute and Olaf Spinczyk from Magdeburg University, and Friedrich Schön from GMD-FIRST for the example and helpful discussions.

References

1. W. Nebel and G. Schumacher. Object-oriented hardware modelling: Where to apply and what are the objects. In *Euro-Dac '96 with Euro-VHDL '96*, 1996. 360
2. J. S. Young, J. MacDonald, M. Shilman, P. H. Tabbara, and A. R. Newton. Design and specification of embedded systems in Java using successive, formal refinement. In *Design Automation Conference (DAC'98)*, 1998. 360

3. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. volume 152 of *Nato ASI Series F*. Springer-Verlag, 1996. 360
4. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and systems*, 16(5):1512–1542, September 1994. 360
5. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, February 1995. 360
6. F. Schön and W. Schröder-Preikschat. On the interrupt-transparent synchronization of interrupt-driven code. Arbeitspapiere der GMD, GMD Forschungszentrum Informatik, 1999. 360, 361, 362